引擎的多线程模型

小侨(王英侨)、nil(陈湘) 发布于 :2021-8-11 18:36

简介

刚接触Messiah引擎的开发人员往往会有以下问题。

1. 引擎的很多函数都有后缀，比如 on_ot，on_rdt等，这些后缀是什么意思?
2. 引擎的整个loop是怎么样的?
3. 我在某个地方能不能调用某个函数?

诸如此类的问题，均和引擎的多线程模型有关。由于Messiah引擎是多线程跨平台的引擎，且聚焦于高效率，引擎开发中有些约定俗成的规则。对此不了解的开发人员就可能写出隐藏bug的代码。此外，引擎本身也在不断迭代，以快速响应项目的需求和不断优化效率。在这个过程中，引擎的多线程模型也相应不断迭代。

本文旨在对引擎多线程运行的机制做一个简单的说明。需要注意的是，引擎以前会、现在也会、将来更是会不断改进。因此本文具有版本局限性，主要阐述2021.1版本的内容。本文也会有之前版本的一些回顾，其目的是为了对照。为了便于理解，本文叙述不一定严谨，一切以代码为准(RTFS)。

物理线程

一个app或可执行程序要运行，其载体是操作系统的进程。在进程中，会有不同作用的线程。

一个Messiah游戏进程中，主要的线程如下。

- Main线程，顾名思义，就是执行c++ main函数的线程。整个游戏世界的入口。
- Game线程，承载游戏主要任务的线程，包括游戏逻辑以及渲染。
- Device线程，主要负责执行图形API，和GPU打交道。
- Compute线程组，若干条线程。不同平台、不同CPU有不同数量的compute线程。这些线程就是工具人，帮Game线程、Device线程打工，在引擎tick loop的不同阶段做不同的工作。引擎的多线程主要体现在compute线程组。
- IO线程组，若干条线程。不同平台、不同CPU有不同数量的IO线程。这些线程专门做IO操作，有时候有些耗时甚长的跨帧的工作，也会扔给IO线程组来做。
- 其他线程，如第三方库、第三方中间件等开的线程。

Main线程

有必要单独说下main线程。main线程的主要作用是发起每一帧。也就是代码中的FireFrame。

传统的单线程游戏引擎往往在main执行以下简化的一个loop。

```
while (true)
{
    // new frame
    collect_input();
    game_logic();
    render();
    wait_next_frame();
}
```

在messiah引擎中，这个loop依然存在，但不仅仅在Main线程中。此外，为了效率，上一帧的一些任务会和下一帧的一些任务交叠。由于叠帧的存在，在引擎的开发中，需要时刻警惕data race的发生。Messiah开发的原则是尽可能不要用锁，用流程保证数据的安全。

Messiah的Main线程主要执行一帧的触发工作，大体流程如下。

```
void on_timer()
{
    bool tick = can_fire_logic_frame();
    bool render = can_fire_render_frame();
    if (tick && render)
```

```
    }
    else if (tick)
    {
        fire_logic_frame();
    }
    else if (render)
    {
        fire_render();
    }
    add_next_timer();
}
```

这里可以看到引擎的逻辑帧和渲染帧是可以分开的。在fire_logic_frame和fire_render()中，Main线程也没干什么活，主要是通知Game线程：~~哦！二点几啦！饮下茶先啦！~~ 醒醒，要搬砖了。

除了触发下一帧，Main线程还有一个工作是执行一些需要在主线程中执行的系统API调用。

需要着重强调的是，这里说的Main线程是指c++ main函数执行的线程，这里有些平台差异性需要说明。

- ios下，Main线程就是app的主线程，但特殊的地方在于，app的main是在开发框架中的，暴露给app开发者的只是一些回调。因此为了比较精确地发起每一帧，以及把逻辑帧和渲染帧分离，在ios下单独开了一条Frame线程，专门负责发起每一帧。Main线程则仅执行ios下需要在主线程执行的API。
- android下，app其实就是java包裹c++。对于游戏引擎，几乎所有东西都在c++，但app的主线程其实是跑在java虚拟机中的ui thread。而负责发起引擎每一帧的线程（所谓的Main）其实是执行NDK c++部分main函数的线程。这里的ui thread是android app的概念，和cocos无关，其实就是执行activity的onCreate等函数的线程。对于需要在app主线程执行的API调用，引擎在c++中通过jni调用java代码后，还需要通过runOnUiThread把要执行的调用post到ui thread中执行。

逻辑视角

上面介绍了引擎相关的物理线程。通过引擎附带的profile工具tracy或者optick可以直观地看到这些线程在不同时段执行的任务。

在引擎开发中，实际上更需要关心当前引擎处于什么阶段，正在执行什么任务。该任务在数据访问上有什么限制等等。

由于引擎的多线程任务分发是建立在boost::asio之上的，引擎不同阶段的任务往往使用不同的dispatcher来分发，如EngineDispatcher、RendererDispatcher等。为了保证同一阶段任务按序执行，会使用asio中的strand，即ot、rdt、rst等。

一个类比

asio相关术语的具体含义可以参阅boost文档。这里尝试做个通俗的解释，以便于理解。

以建筑工地承包商类比，工地上有很多不同的工种，比如砌墙、水工、电工、绿化等等。这就需要不同的包工头（dispatcher）承接对应工种并把任务分发给其带领的施工队。需要注意，工地的不同工种是有一定的依赖关系的，比如说楼都没建起来，水工、电工就可以暂时先歇着了。而同一工种的不同工作也可能是有顺序依赖的，比如混凝土铺楼面，是一层一层做的。（strand保序）对应地，有某些工种之间、或是同一工种的不同工作之间，是可以同时进行的。这是自上而下的视角。

对于在工地搬砖的打工人而言，就是迥异的视角了。有的打工人一直跟着一个包工头，有活就干，没活干就睡觉。（如Device线程之于DeviceDispatcher，Main线程之于EngineDispatcher）有的打工人则是多面手，会不同的工种，但只想打散工。每天看下哪个包工头那有活，就去哪个包工头那报到。（如Game线程之于ObjectDispatcher、RendererDispatcher）

Dispatcher

类似于上面的类比，引擎在每帧的不同阶段，通过不同的dispatcher来分发不同的任务。具体在代码中，就是不同的函数，其函数名往往有不同的后缀，以表明此函数执行时所在的阶段。真正执行这些任务的自然是物理线程。物理线程通过不同的DispatchService来轮询对应的dispatcher分发的任务(asio::poll)。有的物理线程只poll一个dispatcher分发的任务，有的物理线程会poll不同dispatcher分发的任务。

下面是引擎中不同dispatcher的简要说明。

- EngineDispatcher，分发任务到Main线程执行。只能在Main线程执行的API一般通过这个dspatcher分发。底层负责执行的自然是Main线程。函数后缀为on_main。
- ObjectDispatcher，负责的是游戏世界逻辑任务的分发执行，承担着整个游戏世界的更新逻辑。python脚本的执行、物理的模拟、角色动画的更新等等都是在这个阶段执行，有时简称**Tick**阶段，或**ot**阶段。ObjectDispatcher可以分发两类任务，一类是保序的，另一类是并行执行的。前者函数后缀为**on_ot**，后者为**on_par**。ot为object

Culling、FrameGraph的结算等等都是在这个阶段执行，有时简称**Render**阶段，或**rdt**阶段。函数后缀为**on_rdt**。需要注意，这个阶段不少函数名是**Tick_on_rdt**，不要和**ot**的tick混为一谈。**rdt**的任务由Game线程执行。在这个阶段中，FrameGraph的Pass Setup、 Culling、FrameGraph的Pass prepare这三个子阶段的任务会通过RendererExecutive来分发，并且由Game线程和Compute线程共同并行执行。

- RendererExecutive，除了在**rdt**阶段分发一些并行任务，其在非OpenGL下也负责在**encode**阶段分发任务。函数后缀为**on_rcp**，**on_rdp**。rcp是render culling parallel的缩写，rdp是render dispatch parallel的缩写。rcp特别用于culling阶段。**encode**阶段指drawcall就绪后，引擎调用图形API进行encode的阶段。RendererExecutive分发的任务主要由Compute线程负责执行。根据不同阶段，有时候Game线程会协助，有时候Device线程也会来协助。
- DeviceDispatcher，专门分发需要在Device线程执行的任务。函数后缀为**on_dt**。除了在**rdp**中执行的encode操作，其他和图形API相关的操作都在**dt**执行。
- ResourceDispatcher，处理资源相关的任务。函数后缀为**on_rst**。这是一个很轻量级的阶段。目前在某个时间点和**ot**汇合执行。
- FrameDispatcher，专门分发和frame触发相关的任务。ios线程下是Frame线程负责执行，其他平台则是Main线程。FrameDispatcher发起的任务主要检查下一帧的时间点是否到达，由此决定发起下一帧，或是设置触发下次时间点检查的timer。

一帧的不同阶段

上面列出了引擎的各个dispatcher。这些dispatcher在每一帧的不同阶段使用。

下面介绍引擎一帧的不同阶段。我们先从粗略的划分开始，逐步细化，以方便理解。

粗略的划分

最粗略的划分，就是把引擎的多线程暂时忽略，把每一帧切分为3部分。

```
void execute_one_frame()
{
    execute_on_ot();

    execute_on_rdt();

    execute_on_dt();

    trigger_next_frame_on_frame();
}
```

简单理解的话，在**ot**阶段执行脚本逻辑，并更新角色Graph、动画、物理等。

在**rdt**阶段，**ot**停止了，引擎可以进行frustum culling，framegraph中pass的连接和结算，准备draw call的数据等。

在**dt**阶段，引擎调用图形API的接口，创建并更新GPU资源（如贴图、Buffer等）、对draw call进行encode，present等。

然后等待触发下一帧。

ot三阶段

ot也可以划分为三个阶段，early tick，parallel tick和post tick。parallel tick阶段的任务，就是由Game线程和Compute线程共同执行的。

```
void execute_one_frame()
{
    {
        early_tick_on_ot();   // Game

        tick_on_par();         // Game + Compute

        post_tick_on_ot();     //Game
    } //execute_on_ot

    execute_on_rdt();          // Game

    execute_on_dt();          // Device

    trigger_next_frame_on_frame();          // Main
}
```

伪代码如下。

```
void execute_on_rdt()
{
    tick_on_rdt();

    render_on_rdt();
}
```

这个阶段主要的工作在tick_on_rdt，会进行framegraph相关工作和frustum culling等。

render_on_rdt主要是准备好RenderJob。

tick_on_rdt还可以粗略细分为以下阶段。

- FrameGraph执行RenderPass的connect和 tick (Pass Connect & Tick)
- FrameGraph并行执行Pass的setup (Pass Setup)
- 并行执行Frustum Culling (Culling)
- FrameGraph进行Compile (Compile)
- FrameGraph并行执行Pass的prepare (Pass Prepare)
- FrameGraph求解 (FrameGraph Resolve)

Pass Connect & Tick和Pass Setup也可以合并作为FrameGraph Setup阶段。

Compile和Pass Prepare可以视为FrameGraph Compile阶段。

于是，从Pipeline和FrameGraph的视角，为以下四个阶段。

- FrameGraph Setup，各个Pass的渲染数据更新、各Pass相关的连接建立
- Culling，此阶段会把IRenderScenePass相关的draw call数据准备好
- FrameGraph Compile，各个Pass的RT之间的关系建立
- FrameGraph Resolve，各Pass执行顺序确定、其他draw call数据准备好

render_on_rdt粗略可分为两个阶段。

- RenderJob准备
- Create Resource，GPU中创建资源

需要注意，Create Resource阶段和**dt**是有重叠的，因为GPU资源的创建需要调用图形API，而这个是**dt**来负责的。这里视为**rdt**的一个阶段，因为资源创建是由**rdt**这边发起，并且需要等待**dt**完成。为何需要等待？因为Create Resource后，如果下一帧的触发时间点已经到达，那么下一帧的逻辑(**ot**)就已经可以开始了。

这里开始，需要把每帧的loop放进多线程的环境中看待了。让我们先忽略和图形API打交道的**dt**，也即底层的Device线程。**ot**和**rdt**是由Game线程执行的。在Game的视角，每一帧的loop很简单。

```
void work_on_game_thread()
{
    // ......

    {
        execute_on_ot();

        execute_on_rdt();

        wait_next_frame_time_point();
    }

    {
        execute_on_ot();

        execute_on_rdt();

        wait_next_frame_time_point();
    }

   // ......
}
```

现在重新考虑和图形API打交道的Device线程，对其而言，每一帧的loop也很简单。(目前不妨把encode看作**dt**的工作，即OpenGLES3的情况。)

```
void work_on_device_thread()
{
    // ......

    {
        wait_next_frame_on_dt();

        create_resource_on_dt();

        update_resource_on_dt();

        encode_on_dt();

        present_on_dt();
    }

    {
        wait_next_frame_on_dt();

        create_resource_on_dt();

        update_resource_on_dt();

        encode_on_dt();

        present_on_dt();
    }

   // ......
}
```

Device线程专门给GPU发送DrawCall，其DP数据来源自然是Game线程。因此Game和Device线程需要一个同步点，上面用wait_next_frame_on_dt表示。
我们结合Game线程和Device线程的loop，就能得到细化后的execute_on_rdt。

```
void execute_on_rdt()
{
    {
        pass_connect_and_tick_on_rdt();

        pass_setup_on_rdp();

        frustum_culling_on_rcp();

        framegraph_compile_on_rdt();

        pass_prepare_on_rdp();

        framegraph_resolve_on_rdt();
    } // tick_on_rdt

    {
        prepare_renderjob_on_rdt();

        wait_last_present_on_rdt();

        trigger_create_resource_on_rdt();
        // create_resource_on_dt();
        wait_resource_created_on_rdt();

        trigger_next_frame_on_rdt();
```

上面的wait_last_present_on_rdt就是**rdt**和**dt**的一个同步点，也即Game线程和Device线程的同步点。其和wait_next_frame_on_dt呼应。**rdt**准备好这一帧的渲染任务(RenderJob)后，需要等待**dt**上一帧的present结束。然后由**rdt**触发**dt**下一帧的create_resource_on_dt，即wait_next_frame_on_dt 任务达成，**dt**往下走。**dt**完成Create Resource后，通知**rdt**继续往下走。于是Game线程可以开始进行下一帧的逻辑。

这里可以看到Create Resource阶段是**rdt**和**dt**同步的一个时间段，**rdt**和**dt**的数据在这个阶段进行交换是安全的。既然是一个时间区间，那就有起始两个点。在**rdt**视角，就是上面的trigger_create_resource_on_rdt和wait_resource_created_on_rdt。那**dt**也应该有两个点，于是Device Thread的loop修正如下。

```
void loop_on_device_thread()
{
    while (true)
    {
        wait_next_frame_on_dt();

        create_resource_on_dt();

        notify_resource_created_on_dt();

        update_resource_on_dt();

        encode_on_dt();

        present_on_dt();

        notify_presented_on_dt();
    }
}
```

wait_next_frame_on_dt和trigger_create_resource_on_rdt对应，notify_resource_created_on_dt和wait_resource_created_on_rdt对应。此外，present完成也是一个同步点。wait_last_present_on_rdt和notify_presented_on_dt对应。

Game线程的loop

我们把细分的**ot**和**rdt**结合起来，就得到一个比较完成的Game线程的loop。

```
void loop_on_game_thread()
{
    while (true)
    {
        {
            early_tick_on_ot();

            tick_on_par();   // Game + Compute

            post_tick_on_ot();
        }  // execute_on_ot

        if (need_render)
        {
            {
                pass_connect_and_tick_on_rdt();

                pass_setup_on_rdp();    // Game + Compute
            } // FrameGraph Setup

            {
                frustum_culling_on_rcp();  // Game + Compute
            } // Culling

            {
                framegraph_compile_on_rdt();
```

```
            {
                framegraph_resolve_on_rdt();
            } // FrameGraph Resolve
        } // tick_on_rdt

        if (need_render)
        {
            prepare_renderjob_on_rdt();

            wait_last_present_on_rdt();

            trigger_create_resource_on_rdt();
            // create_resource_on_dt();
            wait_resource_created_on_rdt();

            trigger_next_frame_on_rdt();
        } // render_on_rdt
    }
}
```

这里，trigger_next_frame_on_rdt和Main/Frame线程中的fire_logic_frame是对应的。两个都执行后，下一帧才能开始执行。

而need_render，则是由Main/Frame线程中的fire_render来设置的，用于逻辑帧和渲染帧分离。

至此，**ot**、**rdt**的流程已然比较清晰了。是时候再回到**dt**了。

Device线程的loop

让我们重新回顾**dt**的encode。在上述中，我们把encode当作dt执行的一个任务。但其实除了es3，其他图形API都支持多线程encode。提及多线程，自然少不了compute线程组。

和**ot**、**rdt**中可以多线程执行的任务一样，encode也由compute线程组来帮忙执行。

```
void loop_on_device_thread()
{
    while (true)
    {
        wait_next_frame_on_dt();

        create_resource_on_dt();

        notify_resource_created_on_dt();

        update_resource_on_dt();

        encode_on_rdp();    // Device + Compute

        present_on_dt();

        notify_presented_on_dt();
    }
}
```

现在，是时候关爱下compute线程组这个到处打工的苦命人了。

Compute线程的loop

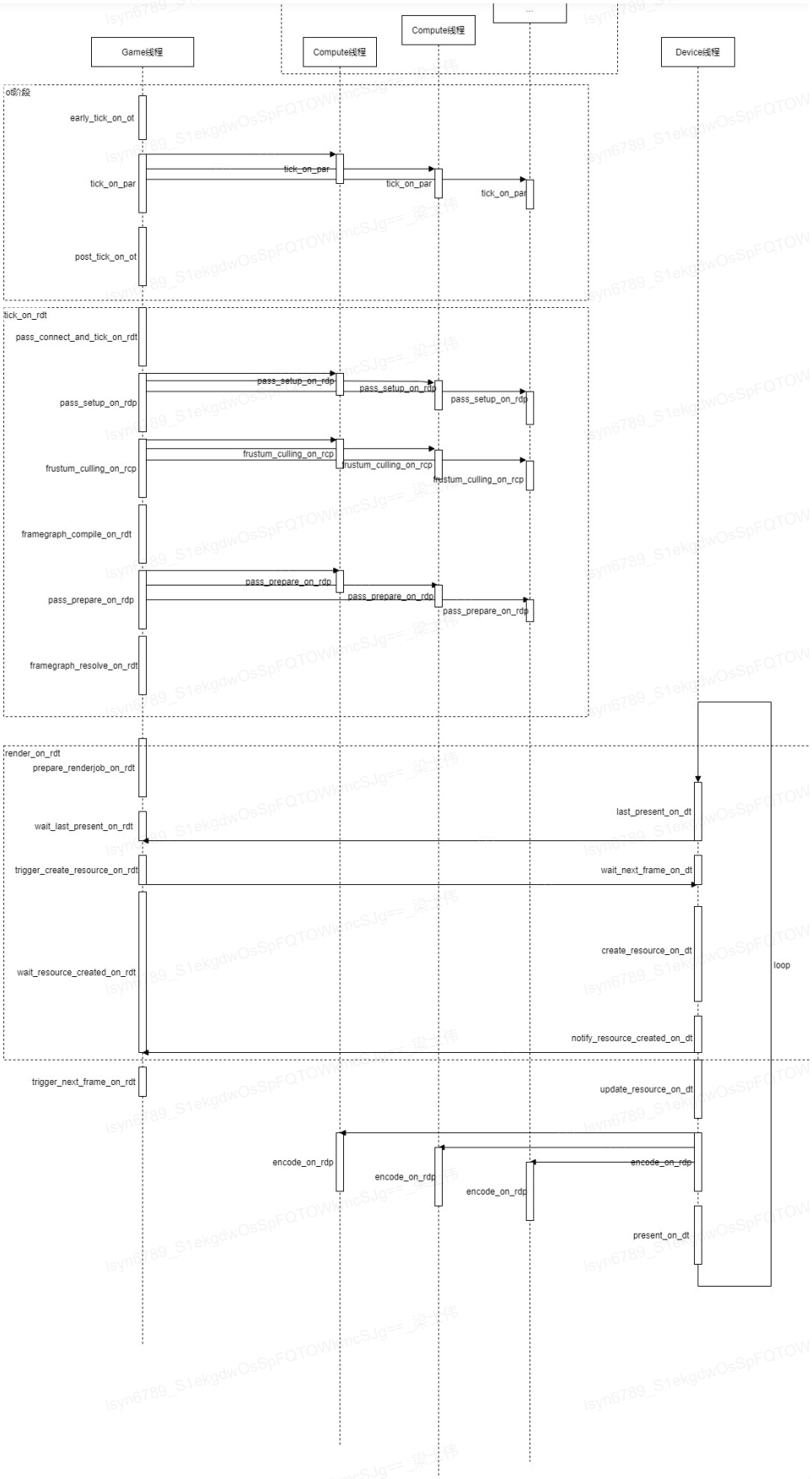
Compute线程的作用就是帮助Game和Device线程。其loop如下：

```
void loop_on_device_thread()
{
    while (true)
    {
        tick_on_par();

        pass_setup_on_rdp();
```

```
        pass_prepare_on_rdp();

        encode_on_rdp();
    }
}
```

上述的每一个任务，都涉及Compute线程组和Game、Device线程的同步。这是由asio和一些atomic量来完成的。具体可参见代码，关键字TDispatcherOnCompleted。

至此，我们大概了解了一帧中各个线程所执行的任务。



版本间差异

本文主要阐述2021.1版本的多线程模型。这里回顾下此前版本的情况，以展示引擎优化迭代的路径。在此后的版本中，引擎会继续迭代，以追求更高的效率。

2020.3

2. Game、Device会和Compute线程组一起干活。

在2020.3，**ot**和**rdt**的任务主要是由Share线程组负责。考虑到**ot**和**rdt**是分时执行的两个阶段，那Share线程组基本上一个时刻只有一个线程在工作。因此2021.1中把Share线程组去掉，替换为一条Game线程。这可以减少线程间切换开销，并且能充分利用手机大中小核架构CPU的大核资源。

在2020.3，**rcp**和**rdp**任务由Compute线程组完成，Share线程组和Device线程袖手旁观。在2021.1中，Game线程、Device线程不再旁观，而是撸起袖子一起干。在加快任务完成之余，也能多利用大核资源。

除了这两点，2021.1也优化了aiso的dispatcher，去除了contexted dispatcher引入的不必要的Task依赖。同时，2021.1也优化了并行Culling，减少了culling任务的分发次数，并且去除了重复的计算。

基本上，用Share线程组替换Game线程，就能得到不严谨的2020.3的多线程模型。

2020.2

版本2020.2和2020.3差异主要集中**rdt**的改动。**rdt**的并行度更高，Culling中一步到位，直接准备好drawcall的数据。**rdt**的整个流程基本都迭代了。

在2020.3中，compute线程组引入，从Share线程组中接手**ot**的**tick_on_par**，也承接了整个Culling的任务，以及多线程encode等等。

在此之前，2020.2中，Share线程组除执行**on_ot**、**on_rdt**的任务，也承接了并行任务的执行，如**tick_on_par**、八叉树**tarversal_on_par**，rdp多线程encode等。2020.2中的Share线程组并不是总是一个线程在工作的。但**rdt**中耗时很多的collect primitive是串行的。因此，2020.3中迭代为并行culling，同时执行collect primitive，并行度更高了。

如上面所提到的，2020.3把所有的并行任务移交给compute线程组，并把share线程数量减少为2。2020.3中没有把share线程数量改为1，是为了保证编辑器逻辑运行正常。在2021.1中，share线程组替换为一条Game线程，波及到的编辑器原来逻辑也进行了调整。

on_any

在2020.2中，有以**on_any**后缀的函数，基本上是CreateResource相关的函数。在dx11中，资源的创建可以在**rdt**，也可以在**dt**，因此以**on_any**为后缀。

在2020.3版本中，所有driver的资源创建都统一在**rdt**的CreateResource阶段，并且引入**ScheduleXXX_on_any**系列函数，把资源的schedule post到**rdt**执行。

这里要提醒，**on_any**是一个有迷惑性的命名，带这后缀的函数从来都不代表在任何时刻、在任何地方都能调用。它只代表这个函数的调用局限在某个阶段，且可能横跨两类线程，一般为**rdt**和**rdp**。它也可能是因为历史原因带此后缀，但目前只能在CreateResource阶段调用。

on_uet

uet是update encode thread的缩写，它代表的是**rdt**和**dt**同步的CreateResource阶段之后，**dt**和**rdp**所处于的UpdateResource和Encode阶段。因此，带此后缀的函数横跨**dt**和**rdp**。

PC编辑器差异

在pc hybrid上，也就是MESSIAH_EDITABLE这个宏生效时，有一个特殊处理。

无论是2021.1之前版本的share线程，还是2021.1的game线程，都只是承担**rdt**的工作，而**ot**的工作由**main**完成。这个特殊的处理是为了方便引擎runtime和编辑器交换数据。在2021.1之后的版本中，这个将会被迭代。

此外，pc release不会有这个特殊处理。在2021.1中，pc hybrid通过命令行启动游戏时，也可以用--object-thread-on-main=0 告知引擎把**ot**放到game线程上。

总结

本文主要叙述引擎2021.1版本的多线程模型，解释了引擎一些函数后缀的含义，希望帮助刚接触引擎的开发人员建立起对引擎loop的初步了解。

多线程的程序开发是困难的，其正确性的保证既需要开发者深入思考、反复review，也需要经历高强度的测试洗礼。任何一个轻率的改动都可能埋下隐患，带来各种偶发的、奇怪的crash。

在此感谢引擎测试同学的有力支持，各种猛烈的极限测试用例是引擎稳定的保障，比如反复跳场景一整天、RenderOption反复切换、数十个ShowRoom同时展示等等。

也感谢项目组的各种反馈，让引擎迭代得更好。

全部评论 10

请输入评论内容

还可以输入 500 个字

(可添加1个视频+5张图片)

匿名 评论

最热 最新

游侠(曹扬)

5楼

要是在5年前有这样的文章，头发就能少掉几根了！！！

2021-08-13 15:12 回复

李强

3楼

侨爷牛逼啊, 解决了我近15年来的困扰!!!

2021-08-12 20:46 回复

zeo(邹晓航)

2楼

侨爷牛逼啊, 解决了我近5年来的困扰!!!

2021-08-12 11:19 回复

千代(徐星)

1楼

侨爷牛逼啊, 解决了我近10年来的困扰!!!

2021-08-12 09:54 回复

小侨(王英侨) 你用引擎还没10年吧。。。

2021-08-12 11:28 作者回复 回复 0

Junyang(邹俊洋)

11楼

侨爷爷牛逼啊，看完我觉得我进化了，值得每日一读

2022-01-13 21:00 回复

孙亚

10楼

好在头发还没有掉多少...

2021-11-15 12:14 回复

Shepard(孙晶)

7楼

anping(万安平)

侨哥牛逼！侨哥英明！信侨哥，无BUG！

2021-09-16 21:11

回复

6楼

一鸥(刘忠源)

要是在3年前有这样的文章，头发就能少掉几根了！！！

2021-08-13 15:39

回复

4楼

Kane(范健锋)

侨爷牛逼

2021-08-13 10:44

回复

加载完毕,没有更多了